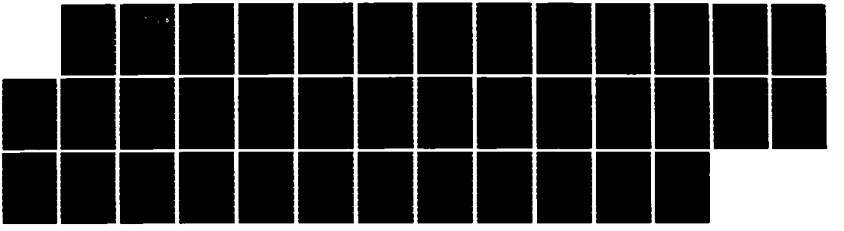
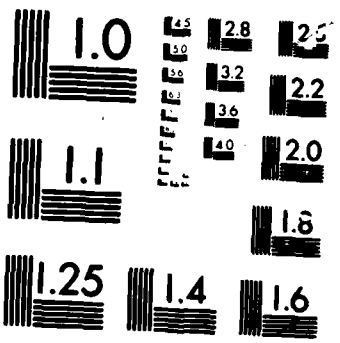
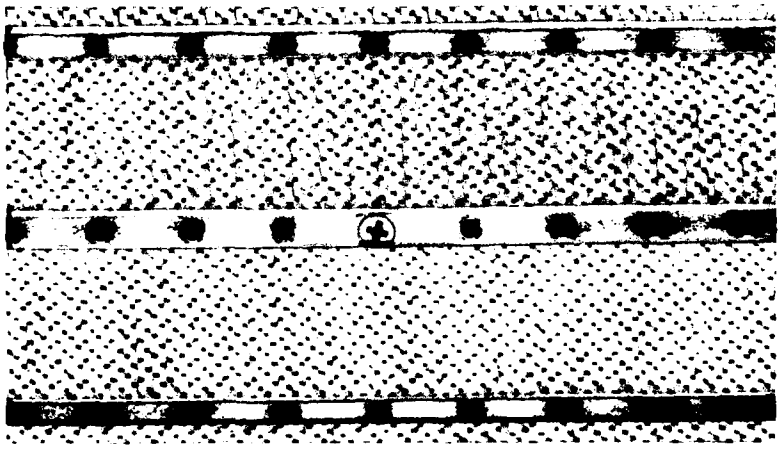
AD-A168 598 HOFF PRELIMINARY DEFINITION VERSION I(1)(U) BROWN UNIV 1/1
PROVIDENCE RI DEPT OF COMPUTER SCIENCE B A DALIO
FEB 86 CS-86-04 83-01-032

UNCLASSIFIED F/G 9/2 NL





BROWN UNIVERSITY



Department

of

Computer Science

(1)

Hoff Preliminary Definition, Version I(1)

Brian A Dalio

Technical Report CS-86-04

February, 1986

Hoff Preliminary Definition, Version I(1)[1]

Brian A Dalio

Department of Computer Science
Brown University
Providence, RI 02912, USA

Abstract

Hoff is a function-level hardware design language. It is the highest level input language for HLDCS (High-Level Device Compilation System). In this preliminary report, a definition of each of the particulars of Hoff is presented. Finally, some complete examples are given to demonstrate the succinctness and power of Hoff.

1. Introduction

Hoff was created to be a function-level hardware design language and the highest level input language for the High-Level Device Compilation System (HLDCS). This technical report is the preliminary definitive reference for the syntax and semantics of the Hoff language, version I(1). A tutorial with an expanded explanation of the implications of Hoff at the hardware level is in preparation.

Following the presentation of Hoff itself we have provided some complete (and non-trivial) examples that demonstrate Hoff's succinctness and expressive power.

Before we can discuss the meaning of Hoff programs, it is necessary for us to review Hoff's view of the world — its model of computation. By this exposition, the reader should get a good grasp of what it means to 'think' in Hoff.[2] It is certainly necessary to do so to write good Hoff programs.

2. Model of Computation

A Hoff 'program'[3] resembles the common notion of a software program in many

[1] This work was supported in part by the Semiconductor Research Corporation under Contract 83-01-032 and in part by a Philips NA Fellowship.

[2] This need is reflected in other programming languages as well. How many times has one seen a program written in, say, Pascal but it was obvious that the programmer was 'thinking' FORTRAN while writing it?

[3] This is not really a good word for this notion, but there doesn't seem to be any other available. Since there are

respects. For example, it may have variables; there are sequencing statements to constructs loops, *etc.*; and it may invoke 'subprograms'. There are, however, differences.

The software model normally assumes a large, random-access memory from which instructions and data are fetched and to which results are stored (usually in a serial manner). The central processor contains the instruction decoding and arithmetic/logical manipulation units and in general directs the functioning of the machine. The physical structure of the machine itself is unaffected by the program that it is currently processing (*i.e.*, there are no configurational decisions that can vary based upon the instruction or data stream).

Since the configuration and capabilities of the processor cannot be altered by the instruction stream, it is necessary that the processor include at all times *any* capabilities that *may* be required at some future time. Also, the data and control paths of the processor must be configured as in general a fashion as possible (or practicable) to anticipate future requirements. Thus, a machine of this sort is termed a *general purpose processor.*

For a Hoff program, however, we must alter this model to correspond more closely to the hardware level we are trying to represent. At this hardware level, the so-called 'program' (actually the specification of the device's function) is known at the time the device is constructed. Thus we needn't include any capabilities that are not explicitly required by that function. The physical structure of the device *is* affected by the 'program' that it will be 'running'. The result is therefore *not* a general purpose processor but a *dedicated hardware device* for the specific function given. This device is, in general, useless for any other function than the one it was designed for.

The differences we alluded to previously now begin to make their presence known. There is a very real possibility of *non*-reusability. In a software environment, the use of a compiler that produces a re-entrant instruction stream allows a piece of code to be used by possibly many threads of execution without interference. At the hardware level, however, such reusability is often not automatic and must be carefully (and painfully) designed in if desired. Quite frequently, because of the expense in performance and complexity of design, it is more convenient to merely duplicate functional units so that each 'thread' may have its own.[4]

Another (and related) difference is that of *parallelism.* Since the device is not a single processor executing instructions but rather a collection of (independent) functional units, it is possible for several of these units to be active at the same time. This simultaneous activity provides the capability of parallel computation provided the Hoff program is so segmented. This is the normal mode of operation for hardware devices and should be taken advantage of whenever possible.

When writing Hoff programs one must constantly be aware of the implications of shared hardware and the possibilities of parallelization. The translator is able to determine certain sorts of each but that is no replacement for the designer being aware of such from the start.

---

some valid parallels that may be drawn, we will use it.

[4] Of course, this decision can be based upon design goals (*i.e.*, if the emphasis is speed, duplication is in order; if the emphasis is size, sharing would be desirable.

Now that Hoff's differences from the software case have been explored, we can move along to Hoff's view of hardware units. Essentially there are two classes of circuits that Hoff may produce: *combinational* and *clocked.* Each is explained more fully in a following paragraph.

Combinational

A combinational circuit is one which contains no feedback loops or memory elements. It is therefore merely an acyclic interconnection of primitive gates or other combinational circuits. Information simply 'flows' through the circuit at its own rate. A delay factor must be computed so that the proper length of time may be waited before making use of the result.

Combinational constructions may be considered to be purely functional (or data-flow) in nature and as such will produce a value when applied to a source. Since they are not clocked, they may not directly be used to produce results for memory elements[5] though they may be applied to any source.

Clocked

A clocked circuit is one which produces a result for a memory element or requires sequencing because of a timing restriction. The former case is directly related to stores to *flags* or *registers* since they require clocking for such operations (all writes to them must be synchronous).

The latter case is more complex. Even though the inherent nature of Hoff programs is to be parallel, there are instances when sequential action is needed (*i.e.,* when one operation must be completed before another is allowed to start). This case occurs most frequently with control flow sequencing. For example, the LOOP statement requires each iteration to be complete before the next one starts.[6] It should be noted, however, that not all types of control flow sequencing cause clocked logic to be utilised. For example, an IF statement which uses a combinational expression to select between two combinational expressions is itself combinational.

The direct relationships between Hoff constructs and hardware elements are covered in more detail in the previously mentioned Hoff tutorial (presently in preparation).

---

[5] Stores to memory elements require clocking to signal precisely when the input is to be stored. Since combinational circuits are not clocked, they require the intervention of another mechanism to provide this clocking signal. An example would be the ASSIGNMENT statement with a flag as its destination. This statement may be used to generate a clock signal for a combinational circuit after the proper delay has been allowed for.

[6] Actually the requirement is that it *seem* as if this is the case. For reasons of efficiency, the system may want to perform parallelizations by, *e.g.,* unrolling a loop to have several iterations in progress at once.

3. Lexical Notions

All legal Hoff programs must at the very least be a sequence of acceptable Hoff tokens. Tokens come in several varieties, each of which is explained in a succeeding paragraph.

Reserved Words

Reserved words are the keywords of the language. They are 'noise' words used to separate and identify the units, statements, *etc.* that make up the program. Reserved words may not be used as identifiers (*i.e.*, they truly are reserved). Reserved words must be entered in lower case as Hoff is case sensitive. The reserved words of Hoff are listed in Table 1.

array	break	by	bus	continue
definitions	else	end	error	flag
for	function	if	integer	join
loop	memory	module	of	record
register	signal	split	string	to
type	uses	waitfor		

Table 1. Reserved Words

Identifiers

Identifiers are names created by the user to identify the various parts or data objects of the program. An identifier may not be the same as one of the reserved words. Identifiers are case sensitive. Even though it is possible to have two identifiers that differ only in the case of the letters (*e.g.*, FROB and frob), it is extremely poor form to do so. Another dangerous habit is to use identifiers that differ from reserved words only in case (*e.g.*, array, which is a reserved word; and Array, which is technically a legal identifier). In a later version of Hoff, this case sensitivity may disappear so don't make use of it now.

Legal identifiers are defined by the regular expression [A-Za-z][A-Za-z_0-9]* which requires an identifier to be a letter followed by arbitrarily many letters, digits, or underscores. There is no preset limit to the length of an identifier and all characters are significant.

Examples

foo Abraxis ADDER_16

Enumeration Constants

Enumerations constants are used to represent the items of an enumeration in expressions. They must come from a space separate from that of normal identifiers to avoid ambiguities. Consequently, enumeration constants are defined to be any normal identifier enclosed within single quote marks ('). The identifier 'inside' an

enumeration constant has the same restrictions imposed on it as a normal identifier does.

Examples

'state_1' 'state_2' 'state_N'

Function Constants

Function constants are used to represent operators and module instantiations. To distinguish them from the normal use of an operator or a module, they are enclosed in back quote marks (`). An operator 'inside' the constant may be any of the legal algorithmic operators as given in Table 5. The module instantiation may be a module name along with any required instantiation parameters.

Examples

operators	'!'	'+'
modules	'foo'	'adder(16)'

Type Constants

Types may be passed as instantiation parameters to modules. Type constants are enclosed in brackets ([ ]) to set them off from surrounding code. Within the brackets may be places any legal type definition.

Examples

[array [BUS_WIDTH] of bus_signals]	[flag]
[foo_type(6,[signal])]	[signal]

Numbers

The legal numbers are defined by the regular expression [0-9a-fA-F]+ which requires a number to be a sequence of one or more digits. The number is interpreted as hexadecimal if it is immediately succeeded by an 'x' or 'X'. If it is succeeded by an 'o' or 'O', it is interpreted as an octal number. Otherwise, it is interpreted as a decimal number. Note that in the hexadecimal case, the number must start with a digit to avoid being misinterpreted as an identifier. If the number would otherwise start with a-f, use 0 as the first digit. This will not affect the number's value but will force a numeric interpretation.

Examples

decimal	42	69
hex	0abx	3febx
octal	377o	105o

Comments

Comments are heralded by three contiguous semicolons (;;;) and are terminated by the physical end of the line. There are no inline comments (or the problems that they cause).

Strings

Strings are sequences of characters enclosed by double quote marks ("). Strings may not span a line boundary. There is currently no mechanism for including a double quote mark inside a string. This may or may not change in the near future.[7]

Multicharacter Tokens

Hoff has a few multicharacter tokens to represent certain operators or specifiers. These tokens must be typed contiguously (*i.e.*, no intervening whitespace) in order to be recognized. The currently existing multicharacter tokens (and their names) are listed in Table 2. This set may change as the language develops further.

Whitespace

Hoff is free form. All whitespace (except inside of strings) is ignored. The only required whitespace is that which is necessary to resolve ambiguous situations. For example, two identifiers in a row must have whitespace in-between or they will be interpreted as a single (but longer) identifier.

4. Data Types and Declarations

Hoff's data types (in line with the tone of the language) are simple and to the point. Though they may seem minimal to some, they are nevertheless quite powerful.

Similar to many of the concepts of Hoff, data types and declarations are divided into two broad classes: Algorithmic (or Run-time) and μAlgorithmic (or Compile-time). The algorithmic side is for objects and structures which correspond to parts of the device which are to be used during its operation. The μalgorithmic objects and structures

||	or	&&	and
==	equal	!=	not equal
<=	less than or equal	..	range
>=	greater than or equal	<-	assignment

Table 2. Multicharacter Tokens

[7] This ambivalence is due to strings only currently being usable as error messages. Having a " in an error message just doesn't seem all that useful. When strings may be used in different constructs, this restriction will in all probability disappear.

are only made use of during the compilation of the device. In the following sections, each of these classes (with its associated notions) is considered.

4.1. Algorithmic Types and Declarations

In this section, we will be looking at the notions of type, type definition, type equivalence, and object declaration for the algorithmic side of Hoff. First, the types themselves.

Primitive Data Types

The two primitive data types are the *flag* and the *signal*. Each is commented upon in a succeeding paragraph. Also included is a description of the *enumeration* construct. Enumerations are not strictly speaking a primitive data type but rather a notational convenience. However, as they are intimately related to the primitive data types, they are explained here.

Flag

A *flag* is a binary digit.[8] It has storage associated with it so its value persists from one moment to the next. Flags may only change state in a synchronous manner. Such synchronicity is relative to the clocking factor which controls the storage area for the flag.

The typemark for a flag is simply the reserved word **flag**.

Signal

A *signal* also represents a binary value but it differs from the flag in that it has *no* storage associated with it. A signal's value is valid only as long as there is a source driving it.

The typemark for a signal is simply the reserved word **signal**.

Enumeration

Another sort of entity which would seem to be a primitive data type (but actually isn't) is the *enumeration*. Enumerations are used when a set of mutually exclusive bit patterns is needed but the exact value of each of those bit patterns is immaterial. A variable may not be of type enumeration since that would give no clue as to its storage policy (*i.e.*, should it have storage (as a flag does) or should it not have storage (as a signal does)).

Enumeration constants may be used in expressions as constants. They are replaced by the assigned bit pattern and are as many bits wide as necessary to represent the number of constants in the enumeration. The name assigned to the enumeration may be used to 'size' arrays (via the *sizeof* function).

The typemark for an enumeration is of the following form.

(*enumid* { , *enumid* }*)

[8] We avoid the more common term *bit* because of the preconceived notions that a user may have about them and how they should work.

where *enumid* complies with the guidelines given in the section *Lexical Notions.*

It is appropriate to comment here on the different viewpoint Hoff has on primitive data types from that of a conventional software programming language. Note that Hoff's main concern is the storage class of an object. The designer must decide if an object is to be persistent (flag) or if it merely acts as a route through which information flows (signal).

This view is epitomised by Hoff's treatment of enumerations. Conventionally, such things are data types in their own right (*i.e.,* an object may be so typed, values of that type exist, operations may be performed on such values and objects). In Hoff, however, enumerations are used in an auxiliary fashion: they provide sizing information and unique binary patterns. There are no objects of an enumeration type — only objects properly sized to contain such.

Structured Data Types

The two structuring constructs are *array* and *record.* Each is considered below.

Array

An *array* is a indexable homogeneous structure. Arrays have a *size* and an *element type.* The size indicates the number of elements that the array contains. These elements are always numbered 0 to size-1 but this restriction may be eliminated if experience warrants it. The element type indicates the type of each of the elements of the array.

There are two array constructs which are so common that they deserve a shorthand notation. The first is an array of flags which may be abbreviated as *register.* The second is an array of signals which may be abbreviated as *bus.*

The typemark for an array is of one the following forms.

array [*constexpr*] **of** *typemark*

register [*constexpr*]

bus [*constexpr*]

The *constexpr* is a μalgorithmic expression which is used to determine the size of the array.

Record

A *record* is an unindexable heterogeneous structure. Records are characterized by their fields each of which has a name and a type. Records are unordered in the sense that when writing a Hoff program the fields are accessed not by specifying an index (as is the case with arrays) but by giving the name of the field desired. However, record types with the same fields though in a different order constitute *different* record types.

The typemark of a record is as follows.

record
declarations
end

The *declarations* are used to specify the fields of the record. The form of declarations is explained below in the *Object Declaration* section. Only algorithmic object declarations may appear in a record typemark.

Type Equivalence and Coercion

Type equivalence in Hoff is by *structure* not by *name*. Two types are equivalent if and only if:

(1) They are both *signal* or both *flag*, or

(2) They are both *record*, they have the same number of fields, and the types of corresponding fields are equivalent, or

(3) They are both *array*, they have the same size, and the element types are equivalent.

Frequently, however, it is not necessary for the types of two objects to match exactly but only that the type of one (called the *source*) may be *coerced* into the other (called the *destination*). This is most often seen across assignment operators or in the case of numerical constants.

A source type may be coerced into a destination type via these rules:

(1) If the source is equivalent to the destination (by the rules given above) then no coercion is necessary.

(2) A *signal* may be coerced to a *flag* and vice-versa.

(3) Source type **array** [*SrcSize*] **of** *SrcElementType* may be coerced into destination type **array** [*DstSize*] **of** *DstElementType* if and only if *SrcElementType* may be coerced into *DstElementType* and *SrcSize* $\leq$ *DstSize*. In the case when *SrcSize* is strictly less than *DstSize*, the source is considered to be padded with nulls on the left (so that numbers will work out properly). If right padding is desired, the user must provide it explicitly.

(4) A source of type **record** *SrcFields* **end** is coercible to a destination of type **record** *DstFields* **end** if and only if the number of fields are equal and the types of the fields are pairwise coercible.

(5) A source of type **record** *FieldName* **:** *FieldType* **;** **end** is coercible to a destination of type *DstType* if and only if *FieldType* is coercible to *DstType*.

(6) A source of type *SrcType* is coercible to a destination of type **record** *FieldName* **:** *FieldType* **;** **end** if and only if *SrcType* is coercible to *FieldType*.

Type Definition and Scope

A type definition assigns a name to a type specified in one of the forms given in the preceding sections. The following syntax is used.

typename { (*params*) } = *typemark* ;

This definition establishes *typename* as a typemark for the typemark given on the right hand side. The optional *params* is a list of μalgorithmic variable declarations which serves as formal parameter list. These variables provide a customization ability at object declaration time. This is covered in more detail in the following section.

Note that the typemark given on the right hand side of a definition must be that of a algorithmic type exclusively. The use of a μalgorithmic typemark is not allowed. To emphasize this restriction, μalgorithmic typemarks are referred to as 'μtypemarks'. μtypemarks[9] are covered in detail in a following section.

Type definitions are associated with a particular defining unit. A type definition's scope stretches from its point of definition through any USESing units. In other words, type definition visibility is textually *dynamic*, not *static*.

Object Declaration and Scope

Any algorithmic object used by a Hoff program must be declared and such use must occur inside the scope of that object's declaration. A declaration has one of the following forms.

objectname : *typemark* ;

objectname : *typename* { (*constexprs*) } ;

In the first case, the given object is created with the specified type. In the second case,[10] the given object is created with the type resulting from binding the formal parameters (given when *typename* itself was defined) to the evaluated values of the *constexprs* and then expanding the associated typemark. There may be no undefined μalgorithmic variables in the *constexprs*. The number and type of *constexprs* must be the same as the number and type of *params* given in the type definition. Outside of this restriction, an expression of any μtype (*i.e.*, μalgorithmic type) may be used.[11]

Note that in this declaration no μtypemarks may be used. There may be no mixing of μalgorithmic and algorithmic types or objects.

There is another type of 'object' declaration which follows a similar syntax. This is assinging of a name to a particular instantiation of a used module. The following form is used.

---

[9] 'μtypemark' is merely the capitalization of 'μtypemark'. The Greek letter *mu* is represented as μ (lower case) and μ (upper case). We use *mu* because it is the first letter of the Greek word $\mu\varepsilon\tau\alpha$ (*meta*, obviously).

[10] Note that the second form is, strictly speaking, included in the first. The *Type Definition* section states that the name assigned to a typemark by a type definition is itself a typemark.

[11] Currently, however, it is useful only to use expressions of μtype **integer** or **type**. Those of μtype **integer** may be used to adjust the sizes of objects and those of μtype **type** may be used to adjust the specific types of objects.

name : *moduleid* { (*constexprs*) } ;

The *name* is the tag by which this particular instantiation of a module is to be referenced later. The *moduleid* must be the same as one given the the USES clause in the module's header. The *constexprs* represent the instantiation parameters for this particular type of module. They *must* be present if the module type requires them. They must match in number and type with the declarations from the used module's header.

Algorithmic object declarations are associated with a particular enclosing structure. The enclosing structures for algorithmic objects are BLOCK and #BLOCK. An object's scope stretches from its point of declaration to the end of its innermost enclosing structure. An object remains visible in nested enclosing structures though *not* in invoked modules. In other words, object visibility is textually *static*, not *dynamic*.

4.2. μAlgorithmic Types (μtypes) and Declarations

In this section, the μtypes (*i.e.*, μalgorithmic types) are explained along with some related notions. The μtypes form a much simpler set than those of the algorithmic class. This is due to their intended use; namely, that of expressing generalizations of algorithmic code.

Data Types

All of the μtypes may be considered to be primitive. There are no structured ones. Each of the μtypes is explained in a following paragraph.

Integer

The *integer* μtype corresponds to the usual notion of a signed integer. Though this definition makes no claim about the range of numbers permitted, the underlying implementation may impose restrictions of its own.

The μtypemark for this μtype is **integer**.

String

Objects of *string* μtype are used for passing messages to instantiated modules or as error messages.

The μtypemark for this μtype is **string**.

Function

Objects of *function* μtype are used to supply specialization information to generic modules. They may be used in algorithmic expressions and are replaced by their values when the expression is fully elaborated.

The μtypemark for this μtype is **function**.

Type

Type objects are used to supply customization information to *USES* units in the form of types. Such objects may be used to create specific instantiations of generic operations.

The μtypemark for this μtype is **type**.

μtype Equivalence

μtype equivalence is easy. Two μtypes are equivalent (*i.e.*, match) if they are the same μtypemark. Thus, *string* matches with and only with *string*, *etc.* There is no μtype coercion possible.

Note that it is not possible to mix typemarks and μtypemarks. The former may only be used with algorithmic constructs and objects, the latter with μalgorithmic ones.

μtype Definition

There is no μtype definition facility. Since there are no complex or structured μtypes, a definition facility would merely be renaming of the simple μtypes and this is not currently seen as being terribly useful.

Object Declaration and Scope

μalgorithmic objects may be declared in one of two places: in the header of a module or at the beginning of a μalgorithmic block. For the former case, see the *Module Unit* part of the *Unit Forms* section. For the latter, the following form is used:

 # objectname : *μtypemark* { = *constexpr* } ;

The optional *constexpr* provides an initializing value for the μalgorithmic variable. It is evaluated in context when the declaration is elaborated. Declarations are processed in the order in which they occur so that variables declared previously (though possibly in the same block) may be used in this expression.

The scope of a μalgorithmic object includes only the body of the module (former declaration case) or the #BLOCK (latter declaration case) declaring it. It is visible in any enclosed enclosing statements but not in any enclosed invoked units. Thus, similar to the algorithmic object case, μalgorithmic object visibility is textually *static*, not *dynamic*.

5. Expressions

Expression evaluation is where the heart of Hoff computation lies. Such computations may take place in one of two forms: Algorithmic or μAlgorithmic. Algorithmic computation corresponds to the functioning of the device, μalgorithmic computation corresponds to the operations necessary to construct the device in the first place.

The μalgorithmic expressions are described first.

5.1. μAlgorithmic Expressions

μalgorithmic expressions may be used in many places (*e.g.*, type definition, μalgorithmic statements) but all uses have one thing in common: they must be completely evaluable when the context in which they appear is elaborated.

Currently, the only μtype which may be manipulated in a μalgorithmic expression is **integer**. The operators and functions that manipulate these μalgorithmic values are all integer-valued themselves. The operators are listed in Table 3 in order of precedence. The lowest precedence items are at the bottom, the highest at the top.

These operators carry the normal connotations for use with integer expressions. The integers are represented internally in two's complement form.

The boolean ones produce 0 for a *false* result and -1 for a *true* result. Note that these results are of type **integer**.

Aside from these operators, there are also some builtin functions to provide useful values. The ones which currently exist are given in Table 4. Note that this list is extremely likely to change as more practice in using Hoff indicates the need for others.

5.2. Algorithmic Expressions

Before discussing algorithmic expressions, it is important to understand the basic items (called *rvalues*) upon which operations may be made. Each of the different classes of rvalues are described in a following section.

precedence	*operator (name)*					
1	!	(invert)	—	(negate)		
2	*	(multiply)	/	(divide)		
3	+	(add)	—	(subtract)		
4	!=	(not equal)	<	(less than)	<=	(less than or equal)
	==	(equal)	>	(greater than)	>=	(greater than or equal)
5	&&	(and)	||	(or)		

Table 3. μalgorithmic expression operators

function	*returns*
log2(x:integer)	integer logarithm to base 2 of x
mod(x:integer,y:integer)	integer remainder from dividing x by y
poweroftwo(x:integer)	-1 if x is a power of 2, 0 otherwise
bitswide(x:integer)	number of bits needed to represent x (unsigned)
power(x:integer,y:integer)	integer exponentiation
sizeof(t:type)	number of elements in enumeration t

Table 4. Builtin μalgorithmic functions

Objects

The first class of rvalues comprises the *objects* and is defined by the following rules.

(1) An object may be an *identifier* in which case it must name a previously declared algorithmic variable. The type of this object is the type the variable was declared to be.

(2) An object may be of the form *object*[*constexpr*]. This represents the *indexing* of an array object of a specific (and known at compile time) entry. The object so indexed into must be declared to be of array type and the constexpr must be within its limits. The type of the ultimate object is the element type of the original array object.

(3) An object may be of the form *object*[*constexpr1* .. *constexpr2*]. This represents the *slicing* of a (known at compile time) subarray out of another. The object so sliced into ***must*** be declared to be of array type and the two constexprs must be within its limits. In addition, *constexpr1* must be less than or equal to *constexpr2*. The type of the ultimate object is array[*constexpr2* - *constexpr1* + 1] of the element type of the original array object. Note that this subarray must be indexed 0 .. *size* - 1, not *constexpr1* .. *constexpr2*.

(4) An object may be of the form *object.id*. This represents the *selection* of a specific field of a record. The object so selected must be of a record type and the given *id* must be one of its defined fields. The type of the ultimate object is the declared type of the field selected.

Function Reference

The next class of rvalues is the *function reference*. This is the mechanism by which a Hoff unit may use previously defined units. A function reference has the following form.

id { (*constexprlist*) } < *runexprlist* >

The *id* corresponds to the name of the unit being invoked. This is the name supplied in its module definition. The list of constexprs must be supplied if the unit invoked requires μalgorithmic instantiation parameters. These expressions are used to tailor the unit to this particular use. The list of runexprs represents the algorithmic information to be passed to the unit.

An alternative form of function reference uses the name assigned to a module instantiation in a declaration. In this case, the

constexprlist is not allowed since the instantiation parameters were supplied at the time the name was assigned.

These two forms of function reference differ in one important respect. In the first case, an abstract entity is being invoked. This implies that if a later function reference happens to invoke the same module even with the same instantiation parameters, it is not certain that the same physical device is being used. Thus, there is the possibility of duplicated hardware.

In the latter case, however, there is no such possibility. The use of the same assigned name implies that the *same* physical device is to be used. In this way, it is possible to create a shared resource and be certain that each user is actually using the same resource.

The type of a function reference is a record of the output values specified in the header of its declaration.

Constants

There are two forms of algorithmic constants in Hoff. The first is a simple numerical constant and the second is an enumeration literal.

Even though there are two written forms, there is in reality only one underlying representation, that of the *bit pattern*. An enumeration literal is converted to a unique representation based upon its position in the enumeration declaration.

The type of a constant is that of an array[*size*] of flag where *size* is exactly big enough to represent the value given. This value will be padded on the left with zeroes[12] if necessary. if necessary for type equivalence. There is *no* automatic truncation of any kind — a type mismatch is reported.

Constructed Records

Sometimes it is necessary to group a set of values together into a single rvalue for more convenient handling or to provide type equivalence. Hoff provides a 'recordizing' process by which a record may be constructed out of a list of algorithmic expressions. The syntax is as follows.

< *runexprlist* >

This construct acts the same as a record composed of the values of the supplied runexprs. However, this construct may not then have a field selected since none of the fields have declared names.

The type of this rvalue is that of a record with (unnamed) fields of the types of the runexprs in the order given.

Constructed Arrays

Akin to the case of constructed records is that of constructed

[12] Note that this would destroy the representation of a negative number. Therefore, it is up to the user to determine the *exact* bit pattern required for any constant and make sure that it is supplied. That's what the 'o' and 'x'

arrays. Sometimes it is necessary to group a set of matching-type values together into a single rvalue for more convenient handling or to provide type equivalence. Hoff provides a 'arrayizing' process by which a array may be constructed out of a list of algorithmic expressions. The syntax is as follows.

[*runexprlist*]

This construct acts the same as a array composed of the values of the supplied runexprs. However, each of the supplied algorithmic expressions must be of the same type as the others.

The type of this rvalue is that of an array with size being the number of runexprs supplied and element type being the type of each of the elements.

Spliced Arrays/Records

Sometimes constructed arrays and records just don't fit the bill when forming structures at runtime. There's one other method called *splicing* which is also available. Splicing works with either a list of records (with any fields) or a list of arrays (with the same element type) but not with both at the same time. The result is a single record or array with all of the components spliced together. This is best explained by example.

If an object of type **array** [5] **of bit** is spliced with an object of **array** [3] **of bit**, the resultant object is of type **array** [8] **of bit**. Note that unlike constructed arrays, only the element type of the component arrays must be the same. Their sizes may vary and the resultant object is of size equal to the sum of the sizes of the component arrays.

In the case of records, the resultant type is a record having as (anonymous) fields all of the fields (in order) of each of the component record types. The following syntax is used for both the record and array cases.

{ *runexprlist* }

Expressions

Algorithmic expressions are used to express the function computed by a Hoff unit. Operations are applied to rvalues (as described in the previous section) and the results may be used in varying ways (*e.g.*, for assignment to an object or as the discriminant in a branch).

The allowed operations are listed in Table 5. Precedence runs from top to bottom (highest to lowest).

---

suffixes on μalgorithmic integer constants are for!

precedence	*operator (name)*			
1	!	(not)		
2	*	(nand)	=	(equivalence)
	^	(exclusive or)	&	(and)
3	|	(or)	+	(nor)

Table 5. Algorithmic operators

These operations are applied on a bit-by-bit basis between rvalues of equivalent type. There is no automatic type coercion outside of that mentioned in the previously appearing *Constants* section.

There are no builtin algorithmic functions. Any functions desired must be defined by the user. Of course, this does not preclude the possibility of standardized libraries for commonly used units (*e.g.*, adders, multipliers of various sizes).

6. Unit Forms

The units of Hoff come in two flavours: *Definitions* and *Module*. Each of these is described in a following paragraph.

Definitions Unit

A definitions unit is used to group a set of related type and constant definitions. Once so grouped, they may be 'invoked' by another definitions or module unit via the USES clause in that unit's header. The syntax of a definitions unit is as follows.

definitions *unitname* { (*params*) } :
    { **uses** (*usedunits*) }
    { *definitions* }+
end

The *unitname* assigns the name which will be used to identify this unit from all others. It is used as the tag when inserting it into the library of available units. Obviously, it must be unique in the system.

The optional *params* provide a customization mechanism which can be used to tailor a definitions unit to specific circumstances. Syntactically, it is a list of μalgorithmic object declarations. Each of these declarations is of the form:

identifier : μ*typemark* ;

When this DEFINITIONS unit is invoked (via a USES clause in some other unit), a corresponding list of constant expressions must be provided. The formals supplied there are bound to the values of the those expressions and may be used (but *not* redefined) by the definitions that follow. The number and type of these constant expressions must match the declarations given in the header of the used unit.

The USES clause allows the semantic 'including' of other definitions units. *Useduniits* is a comma-separated list of unit names and constant expressions. Each element of this list must have the following form.

unitname { (*constexpr* { , *constexpr* }*) }

The definitions from each of the units specified in the USES clause are then made available for the definitions of this unit to use.

The *definitions* are the body of the unit. Each of them may be either a type definition or a constant definition. Type definitions are explained in the *Type Definition* part of the *Data Types and Declarations* section. A constant declaration is of the form:

 # name = constexpr ;

Note that a constant definition may *not* be overridden by a succeeding constant definition.

Module Unit

Module units are the smallest hunks of algorithmic Hoff code that have an identity. They may be invoked by other module units but *not* by definitions units. The syntax of a module unit is as follows.

module *unitname* { (*params*) }
< *inputlist* > :
< *outputlist* >
{ **uses** (*usedunits*) }
block

The *unitname* in this case has the same properties as that of the definitions unit. The same goes for the optional *params*. However, entries in the USES clause are interpreted in a slightly different way. Each item in the USES clause may be one of two things:

(1) The name of a DEFINITIONS unit. In this case the name must be followed by a list of constexprs which supply whatever parameters that particular unit requires. If none are required, the list is omitted.

(2) The name of a MODULE unit. In this case, the name may not be followed by instantiation parameters. Those parameters (if the particular module definition requires them) must be supplied at the time the unit is actually used or assigned to a name.

The *inputlist* and *outputlist* have the same format. The former identifies the inputs to this module, the latter the outputs. Each of these is a list of declarations. The inputs specified may appear only as rvalues and the outputs may appear only as lvalues.

The *block* is described as the BLOCK statement in the following *Algorithmic Statements* section.

7. Statements

Statements come in two flavours based upon when they are acted upon. Each class is discussed in a following section.

μAlgorithmic Statements

μAlgorithmic statements are interpreted during the compilation process at elaboration time. They are used to manipulate statements to form the final unit which is then compiled (*i.e.*, they provide a language in which an algorithm for *creating* the final algorithm may be written). In this respect they may be considered to be part of a 'μlanguage' (hence their name).

μAlgorithmic statement keywords are preceded by '#' to distinguish them from algorithmic statement keywords. This convention indicates the debt that Hoff has towards C in this respect. Each of the μalgorithmic statements of Hoff is presented below with its semantics.

Block

The #BLOCK statement is used both to group statements together as a single statement and also to provide a scope region for μalgorithmic and algorithmic variables. A #BLOCK has the following form.

```
#{
        { μalgorithmic declarations }
        { declarations }
        statements
#}
```

Note that a #BLOCK may be used as a BLOCK (*i.e.*, it's acceptable in places that require an algorithmic statement) but the converse is *not* true.

Assignment

The μalgorithmic #ASSIGNMENT statement is used to set the value of a μalgorithmic variable. The variable supplied on the left-hand side of the statement must be visible. It may not be one of the instantiation parameters supplied to the unit nor may it be a previously declared constant. The expression supplied on the right-hand side of the statement must be a legal μalgorithmic expression of the

same type as the variable. The following form is used.

*variable* = *constexpr* ;

If

The #IF statement is used to select between alternative statements based upon the value of a μalgorithmic integer expression. If the value of the given expression is non-zero, the *if-true* statement is taken; if the value is zero, the *if-false* statement is taken. The **#else** clause is optional. The following syntax is used.

#if (*constexpr*)
if-true statement
{ **#else**
if-false statement }

Loop

The #LOOP statement is used to repetitively evaluate a set of statements. Note that this statement does *not* automatically create a scope for μalgorithmic variables. The loop may only be exited by means of a #BREAK statement (described in a following paragraph).

To provide for later non-ambiguous specification of a particular loop when they are nested, a label may be attached. This label has the same syntax as an identifier, but comes from a different namespace. Thus a label may be the same as an identifier with no ambiguity.

The following form is used for the #LOOP statement.

{ : *id* } **loop**
statements
#end

For

The #FOR statement is used for counted repetition of statement evaluation. Like the #LOOP, a label may be associated with this statement for later identification. Unlike the #LOOP, this statement allows the specification of starting, stopping, and δ values which are applied to the given variable. The body of of the #FOR forms a scope and the supplied index variable is automatically declared (as μtype **integer**) within this scope.

If the δ is positive, then the loop will execute repeatedly until the loop variable is greater than the final expression. If the δ is negative, the loop variable must become less than the final expression. The start, final, and δ expressions are evaluated *once* (before the first iteration of the loop). If the δ is zero, an error results. It is possible for the loop not to be evaluated even once if, *e.g.*, the start

expression is greater than (less than) the final expression and the δ is positive (negative).

If the *δexpr* clause is omitted, the value of the δ is positive one if the final expression is greater than or equal to the start expression and negative one if it is less than the start expression.

The #CONTINUE statement may be used to immediately start the next iteration. The #BREAK statement may be used to immediately terminate the #FOR. See the descriptions of these two statements for more information.

The following syntactic form is used.

{ **:** *id* } **for** (*id* = *startexpr* **to** *finalexpr* { **by** *δexpr* })
    statements
#end

Break

The #BREAK statement is used to terminate the execution of an enclosing #LOOP or #FOR statement. If no particular structure is indicated the innermost enclosing structure is exited. The syntactic form is:

#break { *id* } **;**

Continue

The #CONTINUE statement is used to immediately start the next iteration of a #FOR or #LOOP statement. If none is indicated the innermost enclosing one is continued. The syntax is as follows.

#continue { *id* } **;**

Error

The #ERROR statement is used to signal an error condition detected during μalgorithmic processing. The comma-separated list of constant expressions is optional. If it is present, the string is scanned for occurrences of the escape character (%). The character following the escape character is interpreted as follows:

d, o, x, b
: The value of the corresponding constant expression is printed as a signed decimal (octal, hex, binary) number.

s
: The value of the corresponding identifier is printed as a string.

F
: The value of the corresponding identifier is printed as

an function name (either an operator or module name).

T

The value of the corresponding identifier is printed as a type name.

%

A % is printed.

If there are no constant expressions, the string is printed with no processing.

The proper syntax is:

#error *string* { *constexprlist* } ;

Algorithmic Statements

Algorithmic statements eventually find realization in components, microcode, *etc.* and can affect the execution-time behaviour of the device being designed. Some of these statements have μalgorithmic analogues but there is no chance of ambiguity since algorithmic statement keywords are not preceded by the '#' character.

Each of the algorithmic statements of Hoff is presented below with an explanation of its semantics.

Block

The BLOCK statement is used to group statements together as a single statement. It also provides a scope for any objects declared within it. Such scope stretches from the point of declaration to the end of the block. The following syntactic form is used.

{
    { *declarations* }
    statements
}

Assignment

The ASSIGNMENT statement is used to indicate the transfer of a value from a source to a destination. The right hand side must be an algorithmic expression of a type coercible to that of the left hand side.

The left hand side must be a *lvalue*. Unlike the μalgorithmic assignment, the lhs of the algorithmic assignment may be rather complex. Lvalues are defined by the following rules.

lvalue ::= *object* | < *lvaluelist* >

lvaluelist ::= *lvalue* | *lvaluelist* , *lvalue*

The type of an lvalue is easily determined. For an object, it is merely the type of that object. The type of a list of lvalues is a **record** with the fields being the elements of the list in the order they appear in the list.

The possible complexity of the lvalue allows a *destructuring* assignment similar to that offered in some Lisp implementations. It is useful when many signals are generated in parallel and must be assigned at the same time.

The syntax for an ASSIGNMENT statement is now given.

lvalue <- *runexpr* ;

If

The IF statement is used to select between two alternative paths of execution based upon the value of a algorithmic expression. If the value of the expression is non-zero, the *if-true* branch is taken. If the value of the expression is zero, the *if-false* branch (if any exists) is taken. The following form is used.

if (*runexpr*)
    if-true statement
{ **else**
    if-false statement }

The type of the *runexpr* must be either **flag** or **signal**.

Split

The SPLIT statement is used to conditionally create parallel threads of execution. For each of the included branches, the associated test (which must be of type **flag** or **signal**) is evaluated. If it evaluates to *true* (or if it is omitted entirely), the corresponding statement is executed. Note that all of the tests and statements are evaluated in parallel. When all of the selected statements are finished (or immediately if none were started), execution continues with the statement following the SPLIT.

A label may optionally be attached to this statement. This label may be used to either break out of the statement entirely or to immediately end execution along one of the branches. See the descriptions of the BREAK and SYNC statements, respectively, for more information.

The following syntax is used for the SPLIT statement.

{ : *id* } **split**
    { { (*runexpr*) : } *statement* }+
join

Loop

The LOOP statement is used to repetitively execute a given set of statements. Note that this statement does create a scope. The loop may only be exited by means of a BREAK statement (described in a following paragraph).

To provide for later non-ambiguous specification of a particular loop when they are nested, a label may be attached. This label has the same syntax as an identifier, but comes from a different namespace. Thus a label may be the same as an identifier with no ambiguity.

The following form is used for the LOOP statement.

{ : *id* } **loop**
    statements
end

Waitfor

The WAITFOR statement is used to pause in execution until a given algorithmic expression evaluates to non-zero. This is useful for synchronizing on some anticipated event. The following form is used.

waitfor *runexpr* **;**

The algorithmic expression provided must be of type **signal**.

Break

The BREAK statement is used to terminate the execution of an enclosing LOOP or SPLIT statement. If no particular structure is indicated, the innermost enclosing structure is exited. In the case of a LOOP, control is passed to the succeeding statement. In the case of a SPLIT, the effect is as if all of the branches ceased execution at the same time. However, exactly where in their execution they were is indeterminate. Thus, this ability should be used with great caution.

The syntactic form is:

break { *id* } **;**

Sync

The SYNC statement is used to terminate the execution of a single branch of the indicated SPLIT statement. If no particular statement is indicated, the innermost enclosing branch is synced. Though it may occur anywhere textually inside of the branch, the effect is as if that branch had reached its end. Execution does not proceed from the SPLIT until all of the other branches are finished (or SYNCed) as well.

The correct syntax is:

sync { *id* } ;

8. Examples

In this section, we will be looking at some bits of Hoff code which (it is hoped) will demonstrate Hoff's advantages and power.

Parity Generator

This first example is really quite simple: Design an even parity generator for an input word of *n* bits. The simplicity of this problem statement belies the power of the constructs necessary to accomplish it. The Hoff code for the solution is given in Figure 1.

This module will build a tree of exclusive-or gates to compute the parity. The tree will be as balanced as possible given the size required. Note that this circuit will contain only combinational elements.

```
;;;
;;; parity(SIZE)<input>
;;;     even-parity generator for SIZE-bit bus.
;;;

module parity (SIZE:integer;)
   <word:bus[SIZE];> : <p:signal;>
   uses (parity)
{
   #if (SIZE <= 0)
      #error "parity: illegal SIZE (%d) for input bus" SIZE;
   #else #if (SIZE == 1)
      p <- word[0];
   #else #{
      #LEFTSIZE : integer = SIZE / 2;

      p <- parity(LEFTSIZE)<word[0..LEFTSIZE-1]> ^
          parity(SIZE - LEFTSIZE)<word[LEFTSIZE..SIZE-1]>;
   #}
}
```

Figure 1. Even parity generator

Tree Generator

The previous example built a tree with an exclusive-or gate at each node for the specific case of parity generation. There are other cases where this style of construction

would be useful (*e.g.*, determining if a word is equal to zero (all bits 0)) and to avoid having to write a special case for each one, we can write one general purpose tree generator which will do it for us.

Consider the code given in Figure 2.

```
;;;
;;; treeof(OPR, SIZE, MSG)<input>
;;;        Constructs a tree with SIZE leaves using OPR to
;;;        combine at each level.  MSG is error message to
;;;        display if bad SIZE.
;;;

module treeof(OPR : function; SIZE : integer; MSG : string;)
   <word : bus[SIZE];> : <output : signal;>
   uses(treeof)
{
   #if (SIZE <= 0)
      #error "%s: illegal SIZE (%d) for input bus" MSG, SIZE;
   #else #if (SIZE == 1)
      output <- word[0];
   #else #{
      #LEFTSIZE : integer = SIZE / 2;

      output <- OPR<treeof(OPR,LEFTSIZE,MSG)<word[0..LEFTSIZE-1]>,
                treeof(OPR,SIZE-LEFTSIZE,MSG)<word[LEFTSIZE..SIZE-1]>>;
   #}
}
```

Figure 2. Tree generator

This is simply the parity generator case with the exclusive-or operation replaced by the use of a function passed as specialization information.

Memory Management Unit

Our next example is a bit more ambitious. It is the Hoff representation of a memory management unit for a 68000-style processor. This is not the place for a complete description of the effort which led to this particular design; see [Cook85] for more information.

The complete Hoff code for this example is longer than practical for a figure, so it is given in several pieces in appendices to this paper.

9. Acknowledgements

I am indebted to several persons for quite productive discussions which resulted in Hoff as it is presented in this technical report. Most prominently among those are

William R Cook and John E Savage who helped me through many a rough or confusing spot.

Thanks are also due to William R Cook and Thomas H Freeman for reading this document and offering suggestions for its improvements. All remaining idiocies are my own, of course.

References

Cook85.

COOK, WILLIAM R AND DALIO, BRIAN A, *The Design of a Memory Management Unit: A SLAP/Lucifer Case Study (TR CS-85-03)*, Brown University, Providence, RI (February, 1985).

APPENDICES

1. MMU Definitions Unit
2. MMU Main Module Unit
3. MMU Register Module Unit
4. MMU Mapped OR Unit

```
;;;
;;; memory management unit command definition
;;;

definitions mmu_command :

   commands = record
                reset : signal;
                store : signal;
                retrieve : signal;
                erase : signal;
                write : signal;
                dp : signal;
                vpn : signal;
                pid : signal;
              end;

end
```

```
;;;
;;; memory management unit
;;;
;;; Recursive description of a memory management unit.
;;;  The tree branches UNITS ways at each level. The resulting
;;;  physical page number is ASIZE bits wide (thus ASIZE must be
;;;  an integral multiple of log2(UNITS)). The process id and
;;;  virtual page number are of PIDSIZE and VPNSIZE bits respectively.
;;;

module mmu
   (UNITS:integer; PIDSIZE:integer; VPNSIZE:integer; ASIZE:integer;)
   <pidin:bus[PIDSIZE]; vpnin:bus[VPNSIZE]; cmd:commands; ssi:signal;> :
   <out:bus[ASIZE]; found:signal; sso:signal;>
   uses(mmu_command,mmu_register,mmu,encode,mapor,treeof)
{
   #if (!poweroftwo(UNITS) || UNITS <= 0)
      #error "mmu: UNITS (%d) must be positive power of 2" UNITS;
   #else #if (mod(ASIZE,log2(UNITS)) != 0)
      #error "mmu: ASIZE (%d) must be multiple of log2(UNITS) (%d)"
            ASIZE, log2(UNITS);
   #else #if (ASIZE == log2(UNITS)) #{
      tempss : array [UNITS+1] of signal;
      tempfound : array [UNITS] of signal;

      tempss[0] <- ssi;

      #for (I = 0 to UNITS-1)
         <tempfound[I],tempss[I+1]>
            <- mmu_register(PIDSIZE,VPNSIZE)<pidin,vpnin,cmd,ssi>;
      #end

      sso <- tempss[UNITS];
      out <- encode(UNITS)<tempfound>;
      found <- treeof('|',UNITS,"anyones")<tempfound>;
   #} #else #{
      #A1SIZE : integer = ASIZE - log2(UNITS);

      tempss : array [UNITS+1] of signal;
      tempfound : array [UNITS] of signal;
      tempaddr : array [UNITS] of bus [A1SIZE];

      tempss[0] <- ssi;

      #for (I = 0 to UNITS-1)
         <tempaddr[I],tempfound[I],tempss[I+1]>
            <- mmu(UNITS,PIDSIZE,VPNSIZE,A1SIZE)
                  <pidin,vpnin,cmd,tempss[I]>;
      #end
```

```
        sso <- tempss[UNITS];
        out <- {encode(UNITS)<tempfound>, mapor(UNITS,A1SIZE)<tempaddr>};
        found <- treeof('f',UNITS,"anyones")<tempfound>;
    #}
}
```

```
;;;
;;; memory management unit register
;;;
;;; Base case for recursive memory management unit.  This
;;;  module describes the primitive register which retains
;;;  a process id/virtual page number pair.  The physical
;;;  page number is generated by this unit's position in the
;;;  linear ordering of the leaves of the tree of MMUs.
;;;

module mmu_register(PIDSIZE:integer; VPNSIZE:integer;)
   <pidin:bus[PIDSIZE]; vpnin:bus[VPNSIZE]; cmd:commands; ssi:signal;> :
   <found:signal; sso:signal;>
   uses(mmu_command, treeof)
{
   piddata : register[PIDSIZE];
   vpndata : register[VPNSIZE];
   in_use  : flag;
   dirty   : flag;

   split
      sso <- ssi & in_use;

      (cmd.reset) :
         {
            piddata <- 0;
            vpndata <- 0;
            in_use  <- 0;
            dirty   <- 0;
         }

      (cmd.store & !in_use & ssi) :
         {
            piddata <- pidin;
            vpndata <- vpnin;
            in_use  <- 1;
            dirty   <- 0;
            found   <- 1;
         }

      ( cmd.retrieve & in_use &
        ((cmd.vpn & treeof('&',VPNSIZE,"allones")<vpndata=vpnin>)
             | !cmd.vpn) &
        ((cmd.pid & treeof('&',PIDSIZE,"allones")<piddata=pidin>)
             | !cmd.pid) &
        (cmd.dp & dirty) ) :
         {
            found <- 1;
            dirty <- cmd.write;
         }
```

```
        ( cmd.erase & in_use &
          ((cmd.vpn & treeof('&',VPNSIZE,"allones")<vpndata=vpnin>)
               | !cmd.vpn) &
          ((cmd.pid & treeof('&',PIDSIZE,"allones")<piddata=pidin>)
               | !cmd.pid) &
          (cmd.dp & dirty) ) :
           in_use <- 0;
    join
}
```

```
;;;
;;;    map 'or' across several entities in parallel
;;;

module mapor(NUMBER:integer; SIZE:integer;)
   <input: array[NUMBER] of bus[SIZE];> :
   <output: bus[SIZE];>
   uses(mapor)
{
   #if (NUMBER <= 0)
      #error "mapor: illegal NUMBER (%d)" NUMBER;
   #else #if (NUMBER == 1)
      output <- input[0];
   #else #{
      #LEFTNUMBER : integer = NUMBER / 2;

      output <- mapor(LEFTNUMBER, SIZE)<input[0..LEFTNUMBER-1]> |
            mapor(NUMBER-LEFTNUMBER, SIZE)<input[LEFTNUMBER..NUMBER-1]>;
   #}
}
```

END

DTIC

7-86